A Brief Understanding of README.md

Import Libraries

Code Block

```python
import numpy as np

from common.arguments import parse_args
import torch

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import os
import sys
import errno

from common.camera import *
from common.model import *
from common.loss import *
from common.generators import ChunkedGenerator, UnchunkedGenerator
from time import time
from common.utils import deterministic_random
```

Remarks

- `common.camera` 等引入都是在文件夹`common`下的`.py`文件，例如`common.camera == common/camera.py`。
- 对与torch包中三个函数的名字进行了重载，例如使用`F`来代替`torch.nn.functional`来节省空间。

Load The Dataset

Code Block

```python
args = parse_args()
print(args)

try:
    # Create checkpoint directory if it does not exist
    os.makedirs(args.checkpoint)
except OSError as e:
    if e.errno != errno.EEXIST:
        raise RuntimeError('Unable to create checkpoint directory:', args.checkpoint)

print('Loading dataset...')
dataset_path = 'data/data_3d_' + args.dataset + '.npz'
if args.dataset == 'h36m':
    from common.h36m_dataset import Human36mDataset
    dataset = Human36mDataset(dataset_path)
elif args.dataset.startswith('humaneva'):
    from common.humaneva_dataset import HumanEvaDataset
    dataset = HumanEvaDataset(dataset_path)
elif args.dataset.startswith('custom'):
    from common.custom_dataset import CustomDataset
    dataset = CustomDataset('data/data_2d_' + args.dataset + '_' + args.keypoints + '.npz')
else:
    raise KeyError('Invalid dataset')
```

Remarks

- `Keyerror`在使用映射中不存在的键时引发
- 先尝试新建`./checkpoint`文件夹。如果文件夹已经存在，不再新建；如果还不存在，则新建文件夹。如果不在这两种情况之内，则报错，终止程序。
- 定义变量`dataset_path`为数据集路径变量。观察`data/data_3d`中我们的数据集：
- 根据`README.md`说明，`python run.py -e 80 -k cpn_ft_h36m_dbb -arc 3,3,3,3,3`是成功运行的解，这说明我们并没有人为输入`path`参数，而是在`run.py`中已经定义了这个参数。
- 根据`DATASET.md`说明，目录结构已经符合如下样式：

```
/path/to/dataset/S1/MyPoseFeatures/D3_Positions/Directions 1.cdf
/path/to/dataset/S1/MyPoseFeatures/D3_Positions/Directions.cdf
...
```

- 再根据`DATASET.md`指出的`pyscript`脚本（新版本，不使用`matlab`）

```
cd data
python prepare_data_h36m.py --from-source-cdf /path/to/dataset
cd ..
```

我们看进`prepare_data_h36m.py`的代码：

```
import argparse
import os
import zipfile
import numpy as np
import h5py
from glob import glob
from shutil import rmtree

import sys # 这是system扩展包，用于进行系统的io处理
sys.path.append('../')
from common.h36m_dataset import Human36mDataset # 引入../common中写好的h36m_dataset.py脚本
from common.camera import world_to_camera, project_to_2d, image_coordinates
from common.utils import wrap

output_filename = 'data_3d_h36m'
output_filename_2d = 'data_2d_h36m_gt'
subjects = ['S1', 'S5', 'S6', 'S7', 'S8', 'S9', 'S11']

```

```python
if __name__ == '__main__': # execute only if run as a script
    if os.path.basename(os.getcwd()) != 'data':
        print('This script must be launched from the "data" directory') # 必须在给定的文件夹启动本脚本
        exit(0)

    parser = argparse.ArgumentParser(description='Human3.6M dataset downloader/converter')

    # Convert dataset preprocessed by Martinez et al. in https://github.com/una-dinosauria/3d-pose-baseline
    parser.add_argument('--from-archive', default='', type=str, metavar='PATH', help='convert preprocessed dataset')

    # Convert dataset from original source, using files converted to .mat (the Human3.6M dataset path must be specified manually)
    # This option requires MATLAB to convert files using the provided script
    parser.add_argument('--from-source', default='', type=str, metavar='PATH', help='convert original dataset')

    # Convert dataset from original source, using original .cdf files (the Human3.6M dataset path must be specified manually)
    # This option does not require MATLAB, but the Python library cdflib must be installed
    parser.add_argument('--from-source-cdf', default='', type=str, metavar='PATH', help='convert original dataset')

    args = parser.parse_args()

    if args.from_archive and args.from_source:
        print('Please specify only one argument')
        exit(0)

    if os.path.exists(output_filename + '.npz'):
        print('The dataset already exists at', output_filename + '.npz')
        exit(0)

    if args.from_archive:
        print('Extracting Human3.6M dataset from', args.from_archive)
        with zipfile.ZipFile(args.from_archive, 'r') as archive:
            archive.extractall()

        print('Converting...')
        output = {}
        for subject in subjects:
            output[subject] = {}
            file_list = glob('h36m/' + subject + '/MyPoses/3D_positions/*.h5')
            assert len(file_list) == 30, "Expected 30 files for subject " + subject + ", got " + str(len(file_list))
            for f in file_list:
                action = os.path.splitext(os.path.basename(f))[0]

                if subject == 'S11' and action == 'Directions':
                    continue # Discard corrupted video

                with h5py.File(f) as hf:
                    positions = hf['3D_positions'].value.reshape(32, 3, -1).transpose(2, 0, 1)
                    positions /= 1000 # Meters instead of millimeters
                    output[subject][action] = positions.astype('float32')

        print('Saving...')
        np.savez_compressed(output_filename, positions_3d=output)

        print('Cleaning up...')
        rmtree('h36m')

        print('Done.')

    elif args.from_source:
        print('Converting original Human3.6M dataset from', args.from_source)
        output = {}

        from scipy.io import loadmat

        for subject in subjects:
            output[subject] = {}
            file_list = glob(args.from_source + '/' + subject + '/MyPoseFeatures/D3_Positions/*.cdf.mat')
            assert len(file_list) == 30, "Expected 30 files for subject " + subject + ", got " + str(len(file_list))
            for f in file_list:
                action = os.path.splitext(os.path.splitext(os.path.basename(f))[0])[0]

                if subject == 'S11' and action == 'Directions':
                    continue # Discard corrupted video

                # Use consistent naming convention
                canonical_name = action.replace('TakingPhoto', 'Photo') \
                                       .replace('WalkingDog', 'WalkDog')

                hf = loadmat(f)
                positions = hf['data'][0, 0].reshape(-1, 32, 3)
                positions /= 1000 # Meters instead of millimeters
```

```python
                output[subject][canonical_name] = positions.astype('float32')

        print('Saving...')
        np.savez_compressed(output_filename, positions_3d=output)

        print('Done.')

    elif args.from_source_cdf:
        print('Converting original Human3.6M dataset from', args.from_source_cdf, '(CDF files)')
        output = {}

        import cdflib

        for subject in subjects:
            output[subject] = {}
            file_list = glob(args.from_source_cdf + '/' + subject + '/MyPoseFeatures/D3_Positions/*.cdf')
            assert len(file_list) == 30, "Expected 30 files for subject " + subject + ", got " + str(len(file_list))
            for f in file_list:
                action = os.path.splitext(os.path.basename(f))[0]

                if subject == 'S11' and action == 'Directions':
                    continue # Discard corrupted video

                # Use consistent naming convention
                canonical_name = action.replace('TakingPhoto', 'Photo') \
                                       .replace('WalkingDog', 'WalkDog')

                hf = cdflib.CDF(f)
                positions = hf['Pose'].reshape(-1, 32, 3)
                positions /= 1000 # Meters instead of millimeters
                output[subject][canonical_name] = positions.astype('float32')

        print('Saving...')
        np.savez_compressed(output_filename, positions_3d=output)

        print('Done.')

    else:
        print('Please specify the dataset source')
        exit(0)

    # Create 2D pose file
    print('')
    print('Computing ground-truth 2D poses...')
    dataset = Human36mDataset(output_filename + '.npz') # 使用npz文件存储计算出的2d pose gt文件
    output_2d_poses = {}
    for subject in dataset.subjects():
        output_2d_poses[subject] = {}
        for action in dataset[subject].keys():
            anim = dataset[subject][action]

            positions_2d = []
            for cam in anim['cameras']:
                pos_3d = world_to_camera(anim['positions'], R=cam['orientation'], t=cam['translation'])
                pos_2d = wrap(project_to_2d, pos_3d, cam['intrinsic'], unsqueeze=True)
                pos_2d_pixel_space = image_coordinates(pos_2d, w=cam['res_w'], h=cam['res_h'])
                positions_2d.append(pos_2d_pixel_space.astype('float32'))
            output_2d_poses[subject][action] = positions_2d

    print('Saving...')
    metadata = {
        'num_joints': dataset.skeleton().num_joints(),
        'keypoints_symmetry': [dataset.skeleton().joints_left(), dataset.skeleton().joints_right()]
    }
    np.savez_compressed(output_filename_2d, positions_2d=output_2d_poses, metadata=metadata)

    print('Done.')
```

发现最后所有的data都被装进了一个文件`data/data_3d_h36m.npz`中。我们跟着看`README.md`中剩下来的代码：

```
dataset_path = 'data/data_3d_' + args.dataset + '.npz'
if args.dataset == 'h36m':
    from common.h36m_dataset import Human36mDataset
    dataset = Human36mDataset(dataset_path)
elif args.dataset.startswith('humaneva'):
    from common.humaneva_dataset import HumanEvaDataset
    dataset = HumanEvaDataset(dataset_path)
elif args.dataset.startswith('custom'):
    from common.custom_dataset import CustomDataset
    dataset = CustomDataset('data/data_2d_' + args.dataset + '_' + args.keypoints + '.npz')
else:
    raise KeyError('Invalid dataset')
```

- 发现程序已经选择了`Human3.6mDataset(dataset_path)`进行挂载。

Preparing Data

Code Block

```python
print('Preparing data...')
for subject in dataset.subjects():
    for action in dataset[subject].keys():
        anim = dataset[subject][action]

        if 'positions' in anim:
            positions_3d = []
            for cam in anim['cameras']:
                pos_3d = world_to_camera(anim['positions'], R=cam['orientation'], t=cam['translation'])
                pos_3d[:, 1:] -= pos_3d[:, :1] # Remove global offset, but keep trajectory in first position
                positions_3d.append(pos_3d)
            anim['positions_3d'] = positions_3d
```

Remarks

- 对于数据集中每一个对象都执行准备（预处理）操作。
- 删除全局偏移，但将轨迹保持在第一位置。

Load 2D Detections

Code Block

```python
print('Loading 2D detections...')
keypoints = np.load('data/data_2d_' + args.dataset + '_' + args.keypoints + '.npz', allow_pickle=True)
keypoints_metadata = keypoints['metadata'].item()
keypoints_symmetry = keypoints_metadata['keypoints_symmetry']
kps_left, kps_right = list(keypoints_symmetry[0]), list(keypoints_symmetry[1])
joints_left, joints_right = list(dataset.skeleton().joints_left()), list(dataset.skeleton().joints_right())
keypoints = keypoints['positions_2d'].item()

for subject in dataset.subjects():
    assert subject in keypoints, 'Subject {} is missing from the 2D detections dataset'.format(subject)
    for action in dataset[subject].keys():
        assert action in keypoints[subject], 'Action {} of subject {} is missing from the 2D detections dataset'.format(action, subject)
        if 'positions_3d' not in dataset[subject][action]:
            continue

        for cam_idx in range(len(keypoints[subject][action])):

            # We check for >= instead of == because some videos in H3.6M contain extra frames
            mocap_length = dataset[subject][action]['positions_3d'][cam_idx].shape[0]
            assert keypoints[subject][action][cam_idx].shape[0] >= mocap_length

            if keypoints[subject][action][cam_idx].shape[0] > mocap_length:
                # Shorten sequence
                keypoints[subject][action][cam_idx] = keypoints[subject][action][cam_idx][:mocap_length]

        assert len(keypoints[subject][action]) == len(dataset[subject][action]['positions_3d'])
```

Remarks

- 装载2D探测。根据命令`keypoints = np.load('data/data_2d_' + args.dataset + '_' + args.keypoints + '.npz', allow_pickle=True)`推知

主要的变化是`args.keypoints`，因为custom只在inference in the wild的时候才会用到。对于

Code Block

```
for subject in keypoints.keys():
    for action in keypoints[subject]:
        for cam_idx, kps in enumerate(keypoints[subject][action]):
            # Normalize camera frame
            cam = dataset.cameras()[subject][cam_idx]
            kps[..., :2] = normalize_screen_coordinates(kps[..., :2], w=cam['res_w'], h=cam['res_h'])
            keypoints[subject][action][cam_idx] = kps

subjects_train = args.subjects_train.split(',')
subjects_semi = [] if not args.subjects_unlabeled else args.subjects_unlabeled.split(',')
if not args.render:
    subjects_test = args.subjects_test.split(',')
```

```
else:
    subjects_test = [args.viz_subject]

semi_supervised = len(subjects_semi) > 0
if semi_supervised and not dataset.supports_semi_supervised():
    raise RuntimeError('Semi-supervised training is not implemented for this dataset')
```

Remarks

- `RuntimeError`

Function Fetch()

Code Block

```
def fetch(subjects, action_filter=None, subset=1, parse_3d_poses=True):
    out_poses_3d = []
    out_poses_2d = []
    out_camera_params = []
    for subject in subjects:
        for action in keypoints[subject].keys():
            if action_filter is not None:
                found = False
                for a in action_filter:
                    if action.startswith(a):
                        found = True
                        break
                if not found:
                    continue

            poses_2d = keypoints[subject][action]
            for i in range(len(poses_2d)): # Iterate across cameras
                out_poses_2d.append(poses_2d[i])

            if subject in dataset.cameras():
                cams = dataset.cameras()[subject]
                assert len(cams) == len(poses_2d), 'Camera count mismatch'
                for cam in cams:
                    if 'intrinsic' in cam:
                        out_camera_params.append(cam['intrinsic'])

            if parse_3d_poses and 'positions_3d' in dataset[subject][action]:
                poses_3d = dataset[subject][action]['positions_3d']
                assert len(poses_3d) == len(poses_2d), 'Camera count mismatch'
                for i in range(len(poses_3d)): # Iterate across cameras
                    out_poses_3d.append(poses_3d[i])

    if len(out_camera_params) == 0:
        out_camera_params = None
    if len(out_poses_3d) == 0:
        out_poses_3d = None

    stride = args.downsample
    if subset < 1:
        for i in range(len(out_poses_2d)):
            n_frames = int(round(len(out_poses_2d[i])//stride * subset)*stride)
            start = deterministic_random(0, len(out_poses_2d[i]) - n_frames + 1, str(len(out_poses_2d[i])))
            out_poses_2d[i] = out_poses_2d[i][start:start+n_frames:stride]
            if out_poses_3d is not None:
                out_poses_3d[i] = out_poses_3d[i][start:start+n_frames:stride]
    elif stride > 1:
        # Downsample as requested
        for i in range(len(out_poses_2d)):
            out_poses_2d[i] = out_poses_2d[i][::stride]
            if out_poses_3d is not None:
                out_poses_3d[i] = out_poses_3d[i][::stride]

    return out_camera_params, out_poses_3d, out_poses_2d
```

Remarks

Code Block

```
action_filter = None if args.actions == '*' else args.actions.split(',')
if action_filter is not None:
    print('Selected actions:', action_filter)

cameras_valid, poses_valid, poses_valid_2d = fetch(subjects_test, action_filter)
```

```python

filter_widths = [int(x) for x in args.architecture.split(',')]
if not args.disable_optimizations and not args.dense and args.stride == 1:
    # Use optimized model for single-frame predictions
    model_pos_train = TemporalModelOptimized1f(poses_valid_2d[0].shape[-2], poses_valid_2d[0].shape[-1], dataset.skeleton().num_joints(),
                                filter_widths=filter_widths, causal=args.causal, dropout=args.dropout, channels=args.channels)
else:
    # When incompatible settings are detected (stride > 1, dense filters, or disabled optimization) fall back to normal model
    model_pos_train = TemporalModel(poses_valid_2d[0].shape[-2], poses_valid_2d[0].shape[-1], dataset.skeleton().num_joints(),filter_widths=filter_widths,causal=args.causal,dropout=args.dropout,channels=args.channels,dense=args.dense)
```

Remarks

Code Block

```python
model_pos = TemporalModel(poses_valid_2d[0].shape[-2], poses_valid_2d[0].shape[-1], dataset.skeleton().num_joints(),
                            filter_widths=filter_widths, causal=args.causal, dropout=args.dropout, channels=args.channels,
                            dense=args.dense)

receptive_field = model_pos.receptive_field()
print('INFO: Receptive field: {} frames'.format(receptive_field))
pad = (receptive_field - 1) // 2 # Padding on each side
if args.causal:
    print('INFO: Using causal convolutions')
    causal_shift = pad
else:
    causal_shift = 0

model_params = 0
for parameter in model_pos.parameters():
    model_params += parameter.numel()
print('INFO: Trainable parameter count:', model_params)

if torch.cuda.is_available():
    model_pos = model_pos.cuda()
    model_pos_train = model_pos_train.cuda()
```

```

if args.resume or args.evaluate:
    chk_filename = os.path.join(args.checkpoint, args.resume if args.resume else args.evaluate)
    print('Loading checkpoint', chk_filename)
    checkpoint = torch.load(chk_filename, map_location=lambda storage, loc: storage)
    print('This model was trained for {} epochs'.format(checkpoint['epoch']))
    model_pos_train.load_state_dict(checkpoint['model_pos'])
    model_pos.load_state_dict(checkpoint['model_pos'])

    if args.evaluate and 'model_traj' in checkpoint:
        # Load trajectory model if it contained in the checkpoint (e.g. for inference in the wild)
        model_traj = TemporalModel(poses_valid_2d[0].shape[-2], poses_valid_2d[0].shape[-1], 1,
                            filter_widths=filter_widths, causal=args.causal, dropout=args.dropout, channels=args.channels,
                            dense=args.dense)
        if torch.cuda.is_available():
            model_traj = model_traj.cuda()
        model_traj.load_state_dict(checkpoint['model_traj'])
    else:
        model_traj = None
```

Remarks

Code Block

```
test_generator = UnchunkedGenerator(cameras_valid, poses_valid, poses_valid_2d,
                                    pad=pad, causal_shift=causal_shift, augment=False,
                                    kps_left=kps_left, kps_right=kps_right, joints_left=joints_left, joints_right=joints_right)
print('INFO: Testing on {} frames'.format(test_generator.num_frames()))

if not args.evaluate:
    cameras_train, poses_train, poses_train_2d = fetch(subjects_train, action_filter, subset=args.subset)

    lr = args.learning_rate
    if semi_supervised:
        cameras_semi, _, poses_semi_2d = fetch(subjects_semi, action_filter, parse_3d_poses=False)

        if not args.disable_optimizations and not args.dense and args.stride == 1:
            # Use optimized model for single-frame predictions
            model_traj_train = TemporalModelOptimized1f(poses_valid_2d[0].shape[-2], poses_valid_2d[0].shape[-1], 1,
                    filter_widths=filter_widths, causal=args.causal, dropout=args.dropout, channels=args.channels)
        else:
            # When incompatible settings are detected (stride > 1, dense filters, or disabled optimization) fall back to normal model
            model_traj_train = TemporalModel(poses_valid_2d[0].shape[-2], poses_valid_2d[0].shape[-1], 1,
                    filter_widths=filter_widths, causal=args.causal, dropout=args.dropout, channels=args.channels,
                    dense=args.dense)

        model_traj = TemporalModel(poses_valid_2d[0].shape[-2], poses_valid_2d[0].shape[-1], 1,
                            filter_widths=filter_widths, causal=args.causal, dropout=args.dropout, channels=args.channels,
                            dense=args.dense)
        if torch.cuda.is_available():
            model_traj = model_traj.cuda()
            model_traj_train = model_traj_train.cuda()
        optimizer = optim.Adam(list(model_pos_train.parameters()) + list(model_traj_train.parameters()),
                               lr=lr, amsgrad=True)

        losses_2d_train_unlabeled = []
        losses_2d_train_labeled_eval = []
        losses_2d_train_unlabeled_eval = []
        losses_2d_valid = []

        losses_traj_train = []
        losses_traj_train_eval = []
        losses_traj_valid = []
    else:
        optimizer = optim.Adam(model_pos_train.parameters(), lr=lr, amsgrad=True)

    lr_decay = args.lr_decay

    losses_3d_train = []
    losses_3d_train_eval = []
    losses_3d_valid = []

    epoch = 0
    initial_momentum = 0.1
    final_momentum = 0.001


    train_generator = ChunkedGenerator(args.batch_size//args.stride, cameras_train, poses_train, poses_train_2d, args.stride,
                                       pad=pad, causal_shift=causal_shift, shuffle=True, augment=args.data_augmentation,
                                       kps_left=kps_left, kps_right=kps_right, joints_left=joints_left, joints_right=joints_right)
    train_generator_eval = UnchunkedGenerator(cameras_train, poses_train, poses_train_2d,
                                              pad=pad, causal_shift=causal_shift, augment=False)
    print('INFO: Training on {} frames'.format(train_generator_eval.num_frames()))
    if semi_supervised:
        semi_generator = ChunkedGenerator(args.batch_size//args.stride, cameras_semi, None, poses_semi_2d, args.stride,
                                          pad=pad, causal_shift=causal_shift, shuffle=True,
                                          random_seed=4321, augment=args.data_augmentation,
                                          kps_left=kps_left, kps_right=kps_right, joints_left=joints_left, joints_right=joints_right,
                                          endless=True)
        semi_generator_eval = UnchunkedGenerator(cameras_semi, None, poses_semi_2d,
                                                 pad=pad, causal_shift=causal_shift, augment=False)
        print('INFO: Semi-supervision on {} frames'.format(semi_generator_eval.num_frames()))

    if args.resume:
        epoch = checkpoint['epoch']
        if 'optimizer' in checkpoint and checkpoint['optimizer'] is not None:
            optimizer.load_state_dict(checkpoint['optimizer'])
            train_generator.set_random_state(checkpoint['random_state'])
        else:
            print('WARNING: this checkpoint does not contain an optimizer state. The optimizer will be reinitialized.')

        lr = checkpoint['lr']
        if semi_supervised:
            model_traj_train.load_state_dict(checkpoint['model_traj'])
            model_traj.load_state_dict(checkpoint['model_traj'])
```

```
 semi_generator.set_random_state(checkpoint['random_state_semi'])

    print('** Note: reported losses are averaged over all frames and test-time augmentation is not used here.')
    print('** The final evaluation will be carried out after the last training epoch.')

    # Pos model only
    while epoch < args.epochs:
        start_time = time()
        epoch_loss_3d_train = 0
        epoch_loss_traj_train = 0
        epoch_loss_2d_train_unlabeled = 0
        N = 0
        N_semi = 0
        model_pos_train.train()
        if semi_supervised:
            # Semi-supervised scenario
            model_traj_train.train()
            for (_, batch_3d, batch_2d), (cam_semi, _, batch_2d_semi) in \
                zip(train_generator.next_epoch(), semi_generator.next_epoch()):

                # Fall back to supervised training for the first epoch (to avoid instability)
                skip = epoch < args.warmup

                cam_semi = torch.from_numpy(cam_semi.astype('float32'))
                inputs_3d = torch.from_numpy(batch_3d.astype('float32'))
                if torch.cuda.is_available():
                    cam_semi = cam_semi.cuda()
                    inputs_3d = inputs_3d.cuda()

                inputs_traj = inputs_3d[:, :, :1].clone()
                inputs_3d[:, :, 0] = 0

                # Split point between labeled and unlabeled samples in the batch
                split_idx = inputs_3d.shape[0]

                inputs_2d = torch.from_numpy(batch_2d.astype('float32'))
                inputs_2d_semi = torch.from_numpy(batch_2d_semi.astype('float32'))
                if torch.cuda.is_available():
                    inputs_2d = inputs_2d.cuda()
                    inputs_2d_semi = inputs_2d_semi.cuda()
                inputs_2d_cat =  torch.cat((inputs_2d, inputs_2d_semi), dim=0) if not skip else inputs_2d

                optimizer.zero_grad()

                # Compute 3D poses
                predicted_3d_pos_cat = model_pos_train(inputs_2d_cat)

                loss_3d_pos = mpjpe(predicted_3d_pos_cat[:split_idx], inputs_3d)
                epoch_loss_3d_train += inputs_3d.shape[0]*inputs_3d.shape[1] * loss_3d_pos.item()
                N += inputs_3d.shape[0]*inputs_3d.shape[1]
                loss_total = loss_3d_pos

                # Compute global trajectory
                predicted_traj_cat = model_traj_train(inputs_2d_cat)
                w = 1 / inputs_traj[:, :, :, 2] # Weight inversely proportional to depth
                loss_traj = weighted_mpjpe(predicted_traj_cat[:split_idx], inputs_traj, w)
                epoch_loss_traj_train += inputs_3d.shape[0]*inputs_3d.shape[1] * loss_traj.item()
                assert inputs_traj.shape[0]*inputs_traj.shape[1] == inputs_3d.shape[0]*inputs_3d.shape[1]
                loss_total += loss_traj

                if not skip:
                    # Semi-supervised loss for unlabeled samples
                    predicted_semi = predicted_3d_pos_cat[split_idx:]
                    if pad > 0:
                        target_semi = inputs_2d_semi[:, pad:-pad, :, :2].contiguous()
                    else:
                        target_semi = inputs_2d_semi[:, :, :, :2].contiguous()

                    projection_func = project_to_2d_linear if args.linear_projection else project_to_2d
                    reconstruction_semi = projection_func(predicted_semi + predicted_traj_cat[split_idx:], cam_semi)

                    loss_reconstruction = mpjpe(reconstruction_semi, target_semi) # On 2D poses
                    epoch_loss_2d_train_unlabeled += predicted_semi.shape[0]*predicted_semi.shape[1] * loss_reconstruction.item()
                    if not args.no_proj:
                        loss_total += loss_reconstruction
```

```

          # Bone length term to enforce kinematic constraints
          if args.bone_length_term:
              dists = predicted_3d_pos_cat[:, :, 1:] - predicted_3d_pos_cat[:, :, dataset.skeleton().parents()[1:]]
              bone_lengths = torch.mean(torch.norm(dists, dim=3), dim=1)
              penalty = torch.mean(torch.abs(torch.mean(bone_lengths[:split_idx], dim=0) \
                                          - torch.mean(bone_lengths[split_idx:], dim=0)))
              loss_total += penalty


          N_semi += predicted_semi.shape[0]*predicted_semi.shape[1]
        else:
          N_semi += 1 # To avoid division by zero

        loss_total.backward()

        optimizer.step()
      losses_traj_train.append(epoch_loss_traj_train / N)
      losses_2d_train_unlabeled.append(epoch_loss_2d_train_unlabeled / N_semi)
    else:
      # Regular supervised scenario
      for _, batch_3d, batch_2d in train_generator.next_epoch():
        inputs_3d = torch.from_numpy(batch_3d.astype('float32'))
        inputs_2d = torch.from_numpy(batch_2d.astype('float32'))
        if torch.cuda.is_available():
          inputs_3d = inputs_3d.cuda()
          inputs_2d = inputs_2d.cuda()
        inputs_3d[:, :, 0] = 0

        optimizer.zero_grad()

        # Predict 3D poses
        predicted_3d_pos = model_pos_train(inputs_2d)
        loss_3d_pos = mpjpe(predicted_3d_pos, inputs_3d)
        epoch_loss_3d_train += inputs_3d.shape[0]*inputs_3d.shape[1] * loss_3d_pos.item()
        N += inputs_3d.shape[0]*inputs_3d.shape[1]

        loss_total = loss_3d_pos
        loss_total.backward()

        optimizer.step()

losses_3d_train.append(epoch_loss_3d_train / N)

# End-of-epoch evaluation
with torch.no_grad():
    model_pos.load_state_dict(model_pos_train.state_dict())
    model_pos.eval()
    if semi_supervised:
        model_traj.load_state_dict(model_traj_train.state_dict())
        model_traj.eval()

    epoch_loss_3d_valid = 0
    epoch_loss_traj_valid = 0
    epoch_loss_2d_valid = 0
    N = 0

    if not args.no_eval:
        # Evaluate on test set
        for cam, batch, batch_2d in test_generator.next_epoch():
            inputs_3d = torch.from_numpy(batch.astype('float32'))
            inputs_2d = torch.from_numpy(batch_2d.astype('float32'))
            if torch.cuda.is_available():
                inputs_3d = inputs_3d.cuda()
                inputs_2d = inputs_2d.cuda()
            inputs_traj = inputs_3d[:, :, :1].clone()
            inputs_3d[:, :, 0] = 0

            # Predict 3D poses
            predicted_3d_pos = model_pos(inputs_2d)
            loss_3d_pos = mpjpe(predicted_3d_pos, inputs_3d)
            epoch_loss_3d_valid += inputs_3d.shape[0]*inputs_3d.shape[1] * loss_3d_pos.item()
            N += inputs_3d.shape[0]*inputs_3d.shape[1]

            if semi_supervised:
                cam = torch.from_numpy(cam.astype('float32'))
                if torch.cuda.is_available():
                    cam = cam.cuda()

                predicted_traj = model_traj(inputs_2d)
                loss_traj = mpjpe(predicted_traj, inputs_traj)
                epoch_loss_traj_valid += inputs_traj.shape[0]*inputs_traj.shape[1] * loss_traj.item()
                assert inputs_traj.shape[0]*inputs_traj.shape[1] == inputs_3d.shape[0]*inputs_3d.shape[1]

                if pad > 0:
                    target = inputs_2d[:, pad:-pad, :, :2].contiguous()
                else:
                    target = inputs_2d[:, :, :, :2].contiguous()
                reconstruction = project_to_2d(predicted_3d_pos + predicted_traj, cam)
                loss_reconstruction = mpjpe(reconstruction, target) # On 2D poses
                epoch_loss_2d_valid += reconstruction.shape[0]*reconstruction.shape[1] * loss_reconstruction.item()
                assert reconstruction.shape[0]*reconstruction.shape[1] == inputs_3d.shape[0]*inputs_3d.shape[1]

        losses_3d_valid.append(epoch_loss_3d_valid / N)
        if semi_supervised:
            losses_traj_valid.append(epoch_loss_traj_valid / N)
            losses_2d_valid.append(epoch_loss_2d_valid / N)


        # Evaluate on training set, this time in evaluation mode
        epoch_loss_3d_train_eval = 0
        epoch_loss_traj_train_eval = 0
        epoch_loss_2d_train_labeled_eval = 0
        N = 0
        for cam, batch, batch_2d in train_generator_eval.next_epoch():
            if batch_2d.shape[1] == 0:
                # This can only happen when downsampling the dataset
                continue

            inputs_3d = torch.from_numpy(batch.astype('float32'))
            inputs_2d = torch.from_numpy(batch_2d.astype('float32'))
            if torch.cuda.is_available():
                inputs_3d = inputs_3d.cuda()
                inputs_2d = inputs_2d.cuda()
            inputs_traj = inputs_3d[:, :, :1].clone()
            inputs_3d[:, :, 0] = 0

            # Compute 3D poses
            predicted_3d_pos = model_pos(inputs_2d)
            loss_3d_pos = mpjpe(predicted_3d_pos, inputs_3d)
            epoch_loss_3d_train_eval += inputs_3d.shape[0]*inputs_3d.shape[1] * loss_3d_pos.item()
            N += inputs_3d.shape[0]*inputs_3d.shape[1]

            if semi_supervised:
                cam = torch.from_numpy(cam.astype('float32'))
                if torch.cuda.is_available():
                    cam = cam.cuda()
                predicted_traj = model_traj(inputs_2d)
                loss_traj = mpjpe(predicted_traj, inputs_traj)
                epoch_loss_traj_train_eval += inputs_traj.shape[0]*inputs_traj.shape[1] * loss_traj.item()
                assert inputs_traj.shape[0]*inputs_traj.shape[1] == inputs_3d.shape[0]*inputs_3d.shape[1]

                if pad > 0:
                    target = inputs_2d[:, pad:-pad, :, :2].contiguous()
                else:
                    target = inputs_2d[:, :, :, :2].contiguous()
                reconstruction = project_to_2d(predicted_3d_pos + predicted_traj, cam)
                loss_reconstruction = mpjpe(reconstruction, target)
                epoch_loss_2d_train_labeled_eval += reconstruction.shape[0]*reconstruction.shape[1] * loss_reconstruction.item()
                assert reconstruction.shape[0]*reconstruction.shape[1] == inputs_3d.shape[0]*inputs_3d.shape[1]

        losses_3d_train_eval.append(epoch_loss_3d_train_eval / N)
        if semi_supervised:
            losses_traj_train_eval.append(epoch_loss_traj_train_eval / N)
            losses_2d_train_labeled_eval.append(epoch_loss_2d_train_labeled_eval / N)

        # Evaluate 2D loss on unlabeled training set (in evaluation mode)
        epoch_loss_2d_train_unlabeled_eval = 0
        N_semi = 0
        if semi_supervised:
            for cam, _, batch_2d in semi_generator_eval.next_epoch():
                cam = torch.from_numpy(cam.astype('float32'))
                inputs_2d_semi = torch.from_numpy(batch_2d.astype('float32'))
                if torch.cuda.is_available():
                    cam = cam.cuda()
                    inputs_2d_semi = inputs_2d_semi.cuda()

                predicted_3d_pos_semi = model_pos(inputs_2d_semi)
                predicted_traj_semi = model_traj(inputs_2d_semi)
                if pad > 0:
                    target_semi = inputs_2d_semi[:, pad:-pad, :, :2].contiguous()
                else:
                    target_semi = inputs_2d_semi[:, :, :, :2].contiguous()
                reconstruction_semi = project_to_2d(predicted_3d_pos_semi + predicted_traj_semi, cam)
                loss_reconstruction_semi = mpjpe(reconstruction_semi, target_semi)

                epoch_loss_2d_train_unlabeled_eval += reconstruction_semi.shape[0]*reconstruction_semi.shape[1] \
                                                      * loss_reconstruction_semi.item()
                N_semi += reconstruction_semi.shape[0]*reconstruction_semi.shape[1]
        losses_2d_train_unlabeled_eval.append(epoch_loss_2d_train_unlabeled_eval / N_semi)

elapsed = (time() - start_time)/60

if args.no_eval:
    print('[%d] time %.2f lr %f 3d_train %f' % (
            epoch + 1,
            elapsed,
            lr,
            losses_3d_train[-1] * 1000))
else:
    if semi_supervised:
        print('[%d] time %.2f lr %f 3d_train %f 3d_eval %f traj_eval %f 3d_valid %f '
              'traj_valid %f 2d_train_sup %f 2d_train_unsup %f 2d_valid %f' % (
                epoch + 1,
                elapsed,
                lr,
                losses_3d_train[-1] * 1000,
                losses_3d_train_eval[-1] * 1000,
                losses_traj_train_eval[-1] * 1000,
                losses_3d_valid[-1] * 1000,
                losses_traj_valid[-1] * 1000,
                losses_2d_train_labeled_eval[-1],
                losses_2d_train_unlabeled_eval[-1],
                losses_2d_valid[-1]))
    else:
        print('[%d] time %.2f lr %f 3d_train %f 3d_eval %f 3d_valid %f' % (
                epoch + 1,
                elapsed,
                lr,
                losses_3d_train[-1] * 1000,
                losses_3d_train_eval[-1] * 1000,
                losses_3d_valid[-1]  *1000))

# Decay learning rate exponentially
lr *= lr_decay
for param_group in optimizer.param_groups:
    param_group['lr'] *= lr_decay
epoch += 1

# Decay BatchNorm momentum
momentum = initial_momentum * np.exp(-epoch/args.epochs * np.log(initial_momentum/final_momentum))
model_pos_train.set_bn_momentum(momentum)
if semi_supervised:
    model_traj_train.set_bn_momentum(momentum)

# Save checkpoint if necessary
if epoch % args.checkpoint_frequency == 0:
    chk_path = os.path.join(args.checkpoint, 'epoch_{}.bin'.format(epoch))
    print('Saving checkpoint to', chk_path)

    torch.save({
        'epoch': epoch,
        'lr': lr,
        'random_state': train_generator.random_state(),
        'optimizer': optimizer.state_dict(),
        'model_pos': model_pos_train.state_dict(),
        'model_traj': model_traj_train.state_dict() if semi_supervised else None,
        'random_state_semi': semi_generator.random_state() if semi_supervised else None,
    }, chk_path)

# Save training curves after every epoch, as .png images (if requested)
if args.export_training_curves and epoch > 3:
    if 'matplotlib' not in sys.modules:
        import matplotlib
        matplotlib.use('Agg')
        import matplotlib.pyplot as plt

    plt.figure()
    epoch_x = np.arange(3, len(losses_3d_train)) + 1
    plt.plot(epoch_x, losses_3d_train[3:], '--', color='C0')
    plt.plot(epoch_x, losses_3d_train_eval[3:], color='C0')
    plt.plot(epoch_x, losses_3d_valid[3:], color='C1')
    plt.legend(['3d train', '3d train (eval)', '3d valid (eval)'])
    plt.ylabel('MPJPE (m)')
    plt.xlabel('Epoch')
    plt.xlim((3, epoch))
    plt.savefig(os.path.join(args.checkpoint, 'loss_3d.png'))

    if semi_supervised:
        plt.figure()
        plt.plot(epoch_x, losses_traj_train[3:], '--', color='C0')
        plt.plot(epoch_x, losses_traj_train_eval[3:], color='C0')
        plt.plot(epoch_x, losses_traj_valid[3:], color='C1')
        plt.legend(['traj. train', 'traj. train (eval)', 'traj. valid (eval)'])
        plt.ylabel('Mean distance (m)')
        plt.xlabel('Epoch')
        plt.xlim((3, epoch))
        plt.savefig(os.path.join(args.checkpoint, 'loss_traj.png'))

        plt.figure()
        plt.plot(epoch_x, losses_2d_train_labeled_eval[3:], color='C0')
        plt.plot(epoch_x, losses_2d_train_unlabeled[3:], '--', color='C1')
        plt.plot(epoch_x, losses_2d_train_unlabeled_eval[3:], color='C1')
        plt.plot(epoch_x, losses_2d_valid[3:], color='C2')
        plt.legend(['2d train labeled (eval)', '2d train unlabeled', '2d train unlabeled (eval)', '2d valid (eval)'])
        plt.ylabel('MPJPE (2D)')
        plt.xlabel('Epoch')
        plt.xlim((3, epoch))
        plt.savefig(os.path.join(args.checkpoint, 'loss_2d.png'))
    plt.close('all')
```

Remarks

Evaluate（本部分不要求理解，只要会用就行）

Code Block

```
# Evaluate
def evaluate(test_generator, action=None, return_predictions=False, use_trajectory_model=False):
    epoch_loss_3d_pos = 0
    epoch_loss_3d_pos_procrustes = 0
    epoch_loss_3d_pos_scale = 0
    epoch_loss_3d_vel = 0
    with torch.no_grad():
        if not use_trajectory_model:
            model_pos.eval()
        else:
            model_traj.eval()
        N = 0
        for _, batch, batch_2d in test_generator.next_epoch():
            inputs_2d = torch.from_numpy(batch_2d.astype('float32'))
            if torch.cuda.is_available():
                inputs_2d = inputs_2d.cuda()

            # Positional model
            if not use_trajectory_model:
                predicted_3d_pos = model_pos(inputs_2d)
            else:
                predicted_3d_pos = model_traj(inputs_2d)

            # Test-time augmentation (if enabled)
            if test_generator.augment_enabled():
                # Undo flipping and take average with non-flipped version
                predicted_3d_pos[1, :, :, 0] *= -1
                if not use_trajectory_model:
                    predicted_3d_pos[1, :, joints_left + joints_right] = predicted_3d_pos[1, :, joints_right + joints_left]
                predicted_3d_pos = torch.mean(predicted_3d_pos, dim=0, keepdim=True)

            if return_predictions:
                return predicted_3d_pos.squeeze(0).cpu().numpy()

            inputs_3d = torch.from_numpy(batch.astype('float32'))
            if torch.cuda.is_available():
                inputs_3d = inputs_3d.cuda()
            inputs_3d[:, :, 0] = 0
            if test_generator.augment_enabled():
                inputs_3d = inputs_3d[:1]

            error = mpjpe(predicted_3d_pos, inputs_3d)
            epoch_loss_3d_pos_scale += inputs_3d.shape[0]*inputs_3d.shape[1] * n_mpjpe(predicted_3d_pos, inputs_3d).item()

            epoch_loss_3d_pos += inputs_3d.shape[0]*inputs_3d.shape[1] * error.item()
            N += inputs_3d.shape[0] * inputs_3d.shape[1]

            inputs = inputs_3d.cpu().numpy().reshape(-1, inputs_3d.shape[-2], inputs_3d.shape[-1])
            predicted_3d_pos = predicted_3d_pos.cpu().numpy().reshape(-1, inputs_3d.shape[-2], inputs_3d.shape[-1])

            epoch_loss_3d_pos_procrustes += inputs_3d.shape[0]*inputs_3d.shape[1] * p_mpjpe(predicted_3d_pos, inputs)

            # Compute velocity error
            epoch_loss_3d_vel += inputs_3d.shape[0]*inputs_3d.shape[1] * mean_velocity_error(predicted_3d_pos, inputs)

    if action is None:
        print('----------')
    else:
        print('----'+action+'----')
    e1 = (epoch_loss_3d_pos / N)*1000
    e2 = (epoch_loss_3d_pos_procrustes / N)*1000
    e3 = (epoch_loss_3d_pos_scale / N)*1000
    ev = (epoch_loss_3d_vel / N)*1000
    print('Test time augmentation:', test_generator.augment_enabled())
    print('Protocol #1 Error (MPJPE):', e1, 'mm')
    print('Protocol #2 Error (P-MPJPE):', e2, 'mm')
    print('Protocol #3 Error (N-MPJPE):', e3, 'mm')
    print('Velocity Error (MPJVE):', ev, 'mm')
    print('----------')

    return e1, e2, e3, ev


if args.render:
    print('Rendering...')

    input_keypoints = keypoints[args.viz_subject][args.viz_action][args.viz_camera].copy()
    ground_truth = None
    if args.viz_subject in dataset.subjects() and args.viz_action in dataset[args.viz_subject]:
        if 'positions_3d' in dataset[args.viz_subject][args.viz_action]:
            ground_truth = dataset[args.viz_subject][args.viz_action]['positions_3d'][args.viz_camera].copy()
    if ground_truth is None:
        print('INFO: this action is unlabeled. Ground truth will not be rendered.')

    gen = UnchunkedGenerator(None, None, [input_keypoints],
                             pad=pad, causal_shift=causal_shift, augment=args.test_time_augmentation,
                             kps_left=kps_left, kps_right=kps_right, joints_left=joints_left, joints_right=joints_right)
    prediction = evaluate(gen, return_predictions=True)
    if model_traj is not None and ground_truth is None:
        prediction_traj = evaluate(gen, return_predictions=True, use_trajectory_model=True)
        prediction += prediction_traj

    if args.viz_export is not None:
        print('Exporting joint positions to', args.viz_export)
        # Predictions are in camera space
        np.save(args.viz_export, prediction)

    if args.viz_output is not None:
        if ground_truth is not None:
            # Reapply trajectory
            trajectory = ground_truth[:, :1]
            ground_truth[:, 1:] += trajectory
            prediction += trajectory

        # Invert camera transformation
        cam = dataset.cameras()[args.viz_subject][args.viz_camera]
        if ground_truth is not None:
            prediction = camera_to_world(prediction, R=cam['orientation'], t=cam['translation'])
            ground_truth = camera_to_world(ground_truth, R=cam['orientation'], t=cam['translation'])
        else:
            # If the ground truth is not available, take the camera extrinsic params from a random subject.
            # They are almost the same, and anyway, we only need this for visualization purposes.
            for subject in dataset.cameras():
                if 'orientation' in dataset.cameras()[subject][args.viz_camera]:
                    rot = dataset.cameras()[subject][args.viz_camera]['orientation']
                    break
            prediction = camera_to_world(prediction, R=rot, t=0)
            # We don't have the trajectory, but at least we can rebase the height
            prediction[:, :, 2] -= np.min(prediction[:, :, 2])

        anim_output = {'Reconstruction': prediction}
        if ground_truth is not None and not args.viz_no_ground_truth:
            anim_output['Ground truth'] = ground_truth

        input_keypoints = image_coordinates(input_keypoints[..., :2], w=cam['res_w'], h=cam['res_h'])

        from common.visualization import render_animation
        render_animation(input_keypoints, keypoints_metadata, anim_output,
                         dataset.skeleton(), dataset.fps(), args.viz_bitrate, cam['azimuth'], args.viz_output,
                         limit=args.viz_limit, downsample=args.viz_downsample, size=args.viz_size,
                         input_video_path=args.viz_video, viewport=(cam['res_w'], cam['res_h']),
                         input_video_skip=args.viz_skip)

else:
    print('Evaluating...')
    all_actions = {}
    all_actions_by_subject = {}
    for subject in subjects_test:
        if subject not in all_actions_by_subject:
            all_actions_by_subject[subject] = {}

        for action in dataset[subject].keys():
            action_name = action.split(' ')[0]
            if action_name not in all_actions:
                all_actions[action_name] = []
            if action_name not in all_actions_by_subject[subject]:
                all_actions_by_subject[subject][action_name] = []
            all_actions[action_name].append((subject, action))
            all_actions_by_subject[subject][action_name].append((subject, action))

    def fetch_actions(actions):
        out_poses_3d = []
        out_poses_2d = []

        for subject, action in actions:
            poses_2d = keypoints[subject][action]
            for i in range(len(poses_2d)): # Iterate across cameras
                out_poses_2d.append(poses_2d[i])

            poses_3d = dataset[subject][action]['positions_3d']
            assert len(poses_3d) == len(poses_2d), 'Camera count mismatch'
            for i in range(len(poses_3d)): # Iterate across cameras
                out_poses_3d.append(poses_3d[i])

        stride = args.downsample
        if stride > 1:
            # Downsample as requested
            for i in range(len(out_poses_2d)):
                out_poses_2d[i] = out_poses_2d[i][::stride]
                if out_poses_3d is not None:
                    out_poses_3d[i] = out_poses_3d[i][::stride]

        return out_poses_3d, out_poses_2d

    def run_evaluation(actions, action_filter=None):
        errors_p1 = []
        errors_p2 = []
        errors_p3 = []
        errors_vel = []

        for action_key in actions.keys():
            if action_filter is not None:
                found = False
                for a in action_filter:
                    if action_key.startswith(a):
                        found = True
                        break
                if not found:
                    continue

            poses_act, poses_2d_act = fetch_actions(actions[action_key])
            gen = UnchunkedGenerator(None, poses_act, poses_2d_act,
                                     pad=pad, causal_shift=causal_shift, augment=args.test_time_augmentation,
                                     kps_left=kps_left, kps_right=kps_right, joints_left=joints_left, joints_right=joints_right)
            e1, e2, e3, ev = evaluate(gen, action_key)
            errors_p1.append(e1)
            errors_p2.append(e2)
            errors_p3.append(e3)
            errors_vel.append(ev)

        print('Protocol #1   (MPJPE) action-wise average:', round(np.mean(errors_p1), 1), 'mm')
        print('Protocol #2 (P-MPJPE) action-wise average:', round(np.mean(errors_p2), 1), 'mm')
        print('Protocol #3 (N-MPJPE) action-wise average:', round(np.mean(errors_p3), 1), 'mm')
        print('Velocity      (MPJVE) action-wise average:', round(np.mean(errors_vel), 2), 'mm')

    if not args.by_subject:
        run_evaluation(all_actions, action_filter)
    else:
        for subject in all_actions_by_subject.keys():
            print('Evaluating on subject', subject)
            run_evaluation(all_actions_by_subject[subject], action_filter)
            print('')
```

Remarks